Rigorous Graphical Modelling of Movement in Collective Adaptive Systems

Rigorous graphical modelling of movement in Collective Adaptive Systems

N.Zoń, S. Gilmore, and J. Hillston

Laboratory for Foundations of Computer Science, School of Informatics,
University of Edinburgh, Edinburgh, Scotland
N.Zon@sms.ed.ac.uk

Abstract. Formal modelling provides valuable intellectual tools which can be applied to the problem of analysis and optimisation of systems. In this paper we present a novel software tool which provides a graphical approach to modelling of Collective Adaptive Systems (CAS) with constrained movement. The graphical description is translated into a model that can be analysed to understand the dynamic behaviour of the system. This generated model is expressed in CARMA, a modern feature-rich modelling language designed specifically for modelling CAS. We demonstrate the use of the software tool with an example scenario representing carpooling, in which travellers group together and share a car in order to reach a common destination. This can reduce their travel time and travel costs, whilst also ameliorating traffic congestion by reducing the number of vehicles on the road.

1 Introduction

Formal modelling of system dynamics makes possible the analysis and optimisation of smart city applications, many of which belong to the category of Collective Adaptive Systems (CAS). CAS are collectives of individual components acting and interacting within the context of a common environment. In contrast to systems in which all components have global and perfect knowledge of the whole system, in CAS each component has its own subset of information with the consequence that one component's knowledge might be erroneous or inconsistent with the knowledge of other components. When components are able to change their location in space, errors in knowledge about location can be understood to reflect the component's sensing capabilities, which can be limited in terms of range and accuracy.

Urban transport systems provide a good example of CAS and have been taken as a motivating context for our work. In this setting, systems often contain components whose movement in space is restricted in some way. For example, in bus systems, we can distinguish components that never change their location (bus stops), components whose movement follows a specific path (buses), as well as components that can move without additional restrictions (bus repair service cars, pedestrians). Other urban transport systems (carpooling, trams, bikesharing) also have components subject to one or more movement restrictions.

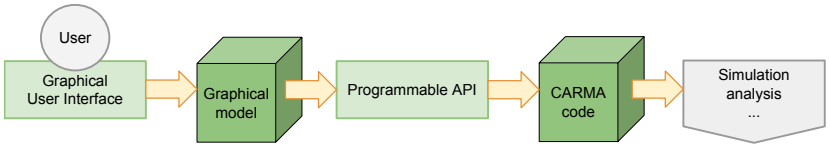

Fig. 1. A flowchart depicting CARMA code generation from graphical input.

Such systems with *constrained movement* are the focus of our work. In these systems the spatial locations of components can have a significant influence on the performance of the collective. A direct influence is observed when an agent is allowed or forbidden to perform specific actions based on the values of their location attributes. An indirect influence is, for example, a situation in which the time taken to traverse a path connecting two points is proportional to the distance between the locations of the two points in space.

CARMA is a formal modelling language designed for the purpose of representing CAS [1]. It provides a syntax for defining components, environments and systems as well as a number of tools for the exploration of the model, such as static analysis and simulation. When an underlying spatial structure also has to be captured by the model, the specification of the environment can become complex and error-prone. Moreover, the amount of CARMA code required for specifying these types of systems grows very rapidly with the complexity of the network and the number of reachable states of each component. In this paper we present an automatic tool for generating the CARMA model code from a graphical input. The tool comprises a Graphical User Interface (GUI) for defining the positions and possible movements of components, as well as a programmable API for the representation and automatic generation of CARMA code.

The GUI supports a newly-developed graphical modelling layer on top of the textual CARMA specification language. Our graphical modelling tool, consisting of a graphical editor and an implementation in the form of an Eclipse IDE plug-in, provides the user with visual ways of representing scenarios involving stationary, mobile and path-restricted agents. The graphical representation is then automatically translated into a CARMA language model template. The code generation scheme is depicted in Fig. 1.

By structuring our contribution in this way, we provide additional flexibility for CARMA users at no extra cost. If a graphical representation of the spatial aspect of the model would be helpful as a communication or documentation aid then the CARMA graphical editor is able to provide it. If, on the other hand, there is no obvious benefit in having a graphical representation for a particular model then the CARMA textual description can be produced directly instead.

The rest of the paper is structured as follows. Section 2 presents background information on CAS, CARMA, and the CARMA tools. Section 3 explains systems with constrained movement and Section 4 presents the graphical represen-

tation of such systems. Section 5 gives more information on the API, and Section 6 presents our case study. We conclude in Section 7.

2 Background

In this section we highlight some of the difficulties encountered when modelling CAS and give an informal introduction to the CARMA specification language. For a more formal definition the reader is referred to [1].

2.1 Modelling CAS

A major issue in faithful representation of CAS is scalability, both with respect to model expression and model analysis. By their nature CAS involve a large number of heterogeneous entities, which are subject to complex rules of interaction and communication but with limited, local knowledge. Furthermore the system is typically highly dynamic with both the entities and the environment subject to change over time. Thus communication based on addresses represented by entity identity or location will fail when entities enter and leave the system and change their location. We choose to use a process algebra-style language in which entities are represented as components and communication is *attribute-based*, meaning that communication partners are selected according to their characteristics rather than their identity or location [2]. The language concerned is CARMA (Collective Adaptive Resource-sharing Markovian Agents), a high-level language designed specifically for modelling CAS [1].

2.2 CARMA semantics

CARMA models consist of a *collective* of components that are situated in the context of an *environment*. Components are the dynamic entities within the model, communicating and collaborating with other components to enact the dynamic behaviour of the system. Each component has an associated *store* recording the current state of *attributes* such as location, or more general status indicators. This captures the local knowledge of the component.

These attributes form the basis of *attribute-based communication* where communication groups are dynamically-formed, making it possible to restrict the communication to sub-groups when it is appropriate to do so. These dynamically-formed communication groups are known as *ensembles* [3]. Examples of restrictions could include only co-located components, only components with adequate security permissions, or only components with sufficient battery charge. Restrictions are expressed as predicates associated with an action, and can be imposed by both the sender and the receiver.

Communication can be asynchronous, non-blocking *broadcast* communication (with many recipients) or synchronous, blocking *unicast* communication α (with only a single recipient). Broadcast communication on name α is denoted by $\alpha^\star$ whereas unicast communication is simply α. Communication occurs in a

CARMA model when an output action from one component is matched with input actions of other components and both predicates are satisfied. Output predicates π place restrictions on the allowable receivers by requiring their local stores to satisfy the predicate. Input predicates similarly place restrictions on the admissible senders but can also inspect the values which are being sent, and might refuse a communication on the basis of these values if they are out-of-range or in some other way erroneous. Values which are accepted can be stored with an update σ. Additionally, process predicates can disallow certain behaviours in a component on the basis of the current state of the store; the process $[\pi]P$ will only evolve to the process P if the predicate π is satisfied.

Processes ($P, Q, \ldots$) in CARMA are thus defined by the following grammar:

$$\begin{aligned} P, Q &::= \mathbf{nil} \mid \mathbf{kill} \mid act.P \mid P + Q \mid P|Q \mid [\pi]P \mid A \quad (A \triangleq P) \\ act &::= \alpha^\star[\pi]\langle \boldsymbol{e} \rangle \sigma \mid \alpha^\star[\pi](\boldsymbol{x})\sigma \mid \alpha[\pi]\langle \boldsymbol{e} \rangle \sigma \mid \alpha[\pi](\boldsymbol{x})\sigma \end{aligned}$$

The action prefix $\alpha^\star[\pi]\langle \boldsymbol{e} \rangle \sigma$ specifies a broadcast output of the values in a vector of expressions $\boldsymbol{e}$. The action prefix $\alpha^\star[\pi](\boldsymbol{x})\sigma$ specifies broadcast input of these values into a vector of variables $\boldsymbol{x}$. The versions without the star are the unicast equivalents.

By convention in a CARMA model activity names begin with a lowercase letter, function and component names begin with a capital letter, and process names are written in all caps. Expressions in the CARMA language (as used in function bodies) are generated by the following grammar.

$$\begin{aligned} e_1, e_2, e_3 &::= \mathbf{return}\ e_1 \mid \mathbf{if}(e_1)\{e_2\} \mid \mathbf{if}(e_1)\{e_2\}\ \mathbf{else}\ \{e_3\} \mid e_1; e_2 \mid a_1 \mid b_1 \\ a_1, a_2 &::= 0 \mid 1 \mid \cdots \mid -a_1 \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2 \\ b_1, b_2 &::= \mathbf{true} \mid \mathbf{false} \mid a_1 > a_2 \mid a_1 >= a_2 \mid a_1 == a_2 \mid a_1 <= a_2 \mid a_1 < a_2 \\ &\quad \mid\ !b_1 \mid b_1\ \&\&\ b_2 \mid b_1 || b_2 \end{aligned}$$

The environment in a CARMA model provides a context for the components. It imposes constraints on activities performed by components, determining the rate at which activities such as communication or movement can take place, with the option to set the rate to zero, if necessary environmental conditions are not met.

CAS are inherently spatially distributed systems and they typically involve large populations of components with the location of a component often constraining the activities that it can perform. In a CARMA model the responsibility for exerting these constraints lies with the environment. Thus the environment records the global state of the model and mediates the component interactions in the collective. Capturing complex spatial arrangements of components can mean that the environment must include functions to represent the spatial structures and the permissible placement and movement of components within those structures. For example, in a recently published CARMA model [4], ambulances travel along paths in a network, in order to reach locations at which accidents have occurred. There are two types of stationary components, hospitals and stations, and these are the locations to which ambulances can return when idle, until being activated when an accident occurs. In this scenario, even a

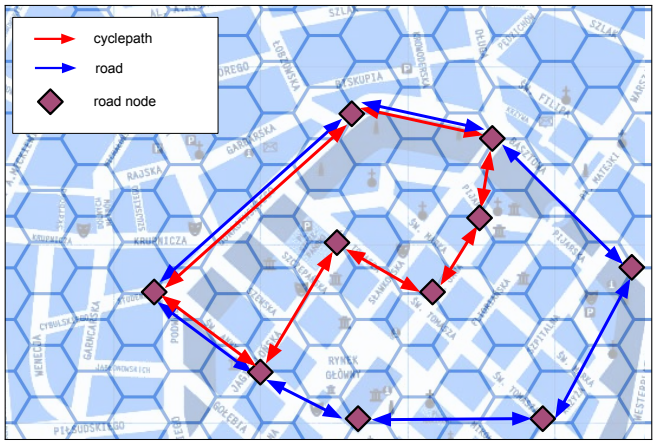

Fig. 2. An example of a system with constrained movement. The two graphs represent cyclepaths (shown in red) and roads (shown in blue). Path nodes are located on nodes of a hexagonal grid (in this case superimposed over a map of the centre of a city), and can be shared between all path types. Path-bounded components can travel along one or both of the defined paths, depending on the component type.

relatively simple road network results in a large amount of CARMA code, in the form of functions in the environment, to capture the spatial layout and possible paths. This is difficult for the modeller and it is this problem which we seek to address with the CARMA graphical editor described in this paper.

2.3 CARMA software and simulation

Software support for modelling in CARMA is provided by the CARMA Eclipse plug-in [5], a toolset which supports the modelling process from model construction to execution and evaluation and analysis of results. Specifically, in this paper we will use the graphical tool for CARMA code generation and a discrete-event stochastic simulator to explore the possible behaviour of the generated CARMA models. Both of the mentioned tools are available for download from the website http://quanticol.sourceforge.net/.

3 Systems with constrained movement

In the current form of the graphical modelling tool we focus on systems in which the movement of components is constrained to follow certain routes in space, each route defined by a path, as seen in Figure 2. More precisely, we consider systems which have the following properties:

1. *The environment of the system contains the definition of one or more paths (represented by graphs) which specific groups of components can traverse in order to change their location.*

2. *Components can be classified into one of three groups based on their ability to move in space:*
   (a) *Stationary components* — their location attributes are constant.
   (b) *Path-bounded components* — can only move along specified paths, their location attribute values belong to the set of node locations of nodes within the specified paths.
   (c) *Free components* — can freely change their location attribute to any value (but are still bound by the environment's definition of space, i.e. a grid).
3. *The spatial locations of components within the system contribute either directly or indirectly to* measures *calculated during model evaluation.*
   – In other words we are interested not only in the topological arrangements of the locations of components but also in the distances between nodes.

Examples of systems with constrained movement include public/private transport networks, heterogeneous computer networks, pedestrian city networks, secure computer networks, animal migration networks, and many others.

4 Graphical representation of spatial elements

In this section we outline the key elements available to the modeller in our graphical editor; essentially these are a graphical palette for specifying paths and a template of icons for representing components.

4.1 Representation of paths

Paths are represented by graphs consisting of nodes, connected by edges. Nodes are placed on a grid which is an unbounded 2D plane, tessellated by hexagons or rectangles to define grid points. To reflect their placement on grid points every node has a location attribute which is a co-ordinate in two-dimensional space. The edges in a path graph are directed and coloured (see Figure 3). The direction of an edge constrains movement on that edge to be in that direction. The colour of an edge constrains the types of components which can move along the edge.

The graphical palette allows the user to instantiate path, and the path connecting them, by laying out the nodes on the hexagonal grid. From the user's point of view, the creation of path node instances is very similar to the creation of component instances. Path nodes are distinct objects from components, and their instances are processed differently for the purpose of CARMA code generation. In contrast to component instances, path nodes are incorporated into CARMA functions instead. Each node can have zero or more incoming and outgoing connections of any colour, each colour representing a distinct path. All nodes have the same colour, and it is assumed that if a node has a connection of a particular colour, any component allowed to move along the route of this colour may assume the location attributes of that node.

Nodes are automatically named by the CARMA graphical editor as they are introduced. A node named nA will have integer x and y coordinates nA_x

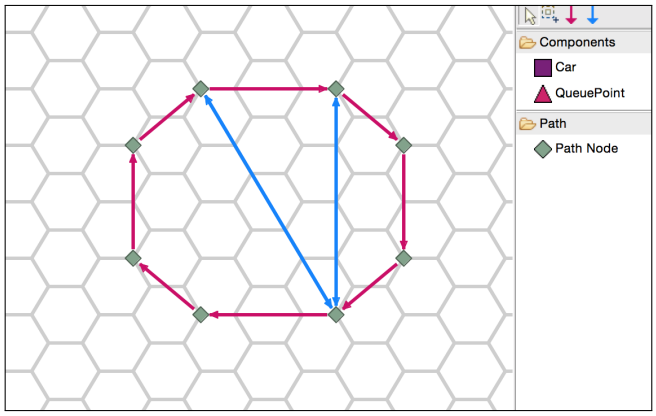

Fig. 3. A screenshot of the graphical interface for path and components layout.

and nA_y. Nodes can later be renamed by the user to semantically-meaningful identifiers.

4.2 Representation of components

The user can specify a component type using structured input. The identifier and appearance of the component can be defined as well as the processes defined in the component, its allowable path and non-movement actions. Once a component type has been defined instances of that component type can then be placed within the graphical layout (by drag and drop). Component instances of the same type differ only in the values of their attributes and therefore can be represented by identical symbols. Their placement on the grid determines their location attribute. The state of a component, given by the value of one of its attributes, can determine if that instance is allowed to move on a particular path. For example, in the carpooling case study presented in Section 6, Car instances that are in the state PRIVILEGED can use both available lanes, while instances in the state NORMAL can only move along the slow lane.

4.3 Example scenarios

Examples of systems that can be defined in the CARMA graphical editor include networks of paths. Each path is specified by a directed graph. The locations of the nodes of these graphs are restricted to a set of points on the plane (i.e. as nodes of a hexagonal grid). Nodes can belong to more than one graph — in this case, a component at a node may have a choice over the available paths, depending on the location, the type of the component, or the state of the instance, as explained above.

A simple urban scenario One example of a scenario with components that have movement constraints is an urban environment with four types of path-bounded components: Bikes, Cars, Pedestrians and Rollerskaters, which move within the environment using paths of the following three types: Pavement, Road, Cyclepath. Components' access to these paths is shown in the table below:

Component name	Pavement	Road	Cyclepath
Bike	allowed	allowed	allowed
Car	forbidden	allowed	forbidden
Pedestrian	allowed	forbidden	forbidden
Rollerskater	allowed	forbidden	allowed

In this example, the ability of a component to move along a path segment of a specific type depends only on the type of the component, not its attribute values.

Listing 1.1 shows an example of an automatically generated function representing a two-way segment of a cyclepath. Similar functions are generated for each path type defined within the system. These functions are used within process predicates to impose the movement constraints that are appropriate for each component type. This can be seen in the subsequent listing, Listing 1.2, showing an automatically generated *Rollerskater* component.

Listing 1.1. A CARMA function to query the existence of a cycle path.

```
fun bool ExistsPath_Cyclepath(int xFrom, int yFrom,
                              int xTo, int yTo){
    if (xFrom == nA_x && yFrom == nA_y
        && xTo == nB_x && yTo == nB_y){
          return true;
        }
    if (xFrom == nB_x && yFrom == nB_x
        && xTo == nA_x && yTo == nA_y){
          return true;
        }
    return false;
}
```

Listing 1.2. The *Rollerskater* component, parameterised by its initial location (x, y), and initial process state Z.

```
component Rollerskater(int x, int y, process Z) {
  store{
    attrib x := x;
    attrib y := y;
  }
  behaviour{
    M =
      [ExistsPath_Cyclepath(my.x, my.y, nA_x, nA_y)]
        move_Cyclepath*[false]⟨⟩{my.x := nA_x, my.y := nA_y}.M
```

```
    + [ExistsPath_Cyclepath(my.x, my.y, nB_x, nB_y)]
        move_Cyclepath*[false]⟨⟩{my.x := nB_x, my.y := nB_y}.M
    + [ExistsPath_Pavement(my.x, my.y, nC_x, nC_y)]
        move_Pavement*[false]⟨⟩{my.x := nC_x, my.y := nC_y}.M;
  }
  init{ Z }
}
```

For each path node accessible from a given path type, we define an action with a predicate which ensures that there exists an incoming connection from the component's current location to the potential next location node. If the predicate is satisfied, the component may perform the action which results in an update of the values of its location attributes.

Movement actions are broadcast output actions, which means that components will perform them spontaneously without trying to synchronize with other components.

The topology-defining functions generated from the layout palette can be seen to have three roles: to store information, provide a mechanism for retrieving it, and to guard the global knowledge with respect to access rules defined for each component and location. In the modelling style implemented in CARMA, only the environment has global knowledge and components have only local knowledge. Thus the components can only access information about the paths in the system through the interface defined by the functions. These functions can be considered to be part of the environment. At the same time, the actions of any component are generated in such a way that only information concerning its current location can be requested. Thus, the restriction that components have only local knowledge is respected. Having no memory of their previous locations and no insight into future ones, components are unable to request information outside of their locality, even though a declarative specification of the network topology is always available to them through the interface.

5 Automatic code generation

The Java API for automatic code generation can be used as part of the Eclipse IDE plug-in as a middle layer between graphical input and the CARMA code input. This API can also be used as a standalone Java package, for users to define models directly from the level of the Java language or to provide their own GUI implementations. One reason to use the API in this way could be if we are generating CARMA code from available runtime data, instead of constructing the graphical representation manually using the graphical editor.

The Java representation of a CARMA model is a two-part specification, consisting of definitions of component types, constants, functions and measures (the template), and their use in a particular case (the instance). The CARMA code generation API reflects this structure, but constrained movement functions and component actions (which usually belong to the template part of a system),

are generated with the use of information about a particular system instance (locations of path nodes and components).

Because of the need to explicitly specify location values and allowable connections for each state of each component, specification of movement constraints in CARMA is error-prone when typed by hand. The automatically generated code can be seen as a draft of a model, providing the definition of the movement policies applicable to a particular scenario. The user can later supplement the code with custom behaviour, where required.

6 Case Study: Carpooling

Carpooling is a means of improving traffic congestion in urban areas where large numbers of people move from one place to another at similar times during the day. It takes advantage of the infrastructure of High Occupancy Vehicle (HOV) road lanes, introduced on main roads in some cities. These lanes are less congested, and therefore allow faster and more comfortable travel; however only cars having at least a particular number of passengers are allowed to use them. The introduction of this infrastructure triggers the spontaneous formation of queue points, where people wait to be picked up by a car travelling in a particular direction. Both the owner of the car and passengers benefit from such an arrangement, saving time and money for journeys. The overall traffic situation in the city also improves since more people will choose to leave their cars at home and become a passenger, therefore reducing the total number of cars on the road.

Modelling carpooling can provide insights into the functioning of the system in practice and inform decisions on where to put pickup points in the network.

In our model, the Car component can change its state between NORMAL and PRIVILEGED, and its ability to move along a certain path depends on the current state of a particular instance (see Listing 1.3).

Component name	Fast lane	Slow lane
Car (NORMAL)	forbidden	allowed
Car (PRIVILEGED)	allowed	allowed

Listing 1.3. A *Car* component which has two local process states, NORMAL and PRIVILEGED. Note that here we show only the movement aspects of behaviour generated from the GUI.

```
component Car(int x, int y, process Z) {
  store{
    attrib x := x;
    attrib y := y;
  }
  behaviour{
    NORMAL =
      [ExistsPath_SlowLane(my.x, my.y, nA_x, nA_y)]
        move_SlowLane*[false]⟨⟩{my.x := nA_x, my.y := nA_y}.NORMAL
```

$$
\begin{array}{l}
\quad + [\mathit{ExistsPath\_SlowLane}(\mathbf{my}.x,\ \mathbf{my}.y,\ nB_x,\ nB_y)] \\
\qquad \mathit{move\_SlowLane}^{\star}[\mathbf{false}]\langle\rangle\{\mathbf{my}.x := nB_x,\ \mathbf{my}.y := nB_y\}.\mathit{NORMAL}; \\
\quad // \text{ Modeller−specified code to be added here.} \\
\quad \mathit{PRIVILEGED} = \\
\qquad [\mathit{ExistsPath\_SlowLane}(\mathbf{my}.x,\ \mathbf{my}.y,\ nA_x,\ nA_y)] \\
\qquad \mathit{move\_SlowLane}^{\star}[\mathbf{false}]\langle\rangle\{\mathbf{my}.x := nA_x,\ \mathbf{my}.y := nA_y\}.\mathit{PRIVILEGED} \\
\quad + [\mathit{ExistsPath\_SlowLane}(\mathbf{my}.x,\ \mathbf{my}.y,\ nB_x,\ nB_y)] \\
\qquad \mathit{move\_SlowLane}^{\star}[\mathbf{false}]\langle\rangle\{\mathbf{my}.x := nB_x,\ \mathbf{my}.y := nB_y\}.\mathit{PRIVILEGED} \\
\quad + [\mathit{ExistsPath\_FastLane}(\mathbf{my}.x,\ \mathbf{my}.y,\ nA_x,\ nA_y)] \\
\qquad \mathit{move\_FastLane}^{\star}[\mathbf{false}]\langle\rangle\{\mathbf{my}.x := nA_x,\ \mathbf{my}.y := nA_y\}.\mathit{PRIVILEGED} \\
\quad + [\mathit{ExistsPath\_FastLane}(\mathbf{my}.x,\ \mathbf{my}.y,\ nB_x,\ nB_y)] \\
\qquad \mathit{move\_FastLane}^{\star}[\mathbf{false}]\langle\rangle\{\mathbf{my}.x := nB_x,\ \mathbf{my}.y := nB_y\}.\mathit{PRIVILEGED}; \\
\quad \} \\
\quad \mathbf{init}\{\ Z\ \} \\
\}
\end{array}
$$

Our model of carpooling is different from the one discussed by Yang and Huang in [6] in that they focus on exploring the various ways in which introducing multiple HOV lanes with toll differentiation influences the overall social welfare in a community, whereas we study the impact of lane speed differentials and the efficiency of passenger loading at queue points. Another approach to the problem was taken by Hussain *et al.* in [7] where they analysed the ways in which potential passengers can negotiate and reach agreements in order to form successful carpools with highest possible levels of satisfaction depending on their preferred start and offload location. Agent-based methods are also used by Guo *et al.* in [8] when using a genetic algorithm to solve the long-term car pooling problem efficiently with limited exploration of the search space. Simulation is the preferred computational method for car-pooling problems because the often-studied long-term car-pooling problem is a computationally hard combinatorial analysis problem best addressed by heuristics and simulation methods [9].

6.1 Specification in CARMA

In CARMA, we are able to define the Carpooling scenario, by specifying the actions available for each component state separately, and relating them to the predefined paths between nodes. In our model of the carpooling scenario, *Car* components move along path segments and can pick up passengers waiting at *QueuePoints* located at path nodes (see Figure 4). The maximum number of passengers that can travel in a car at a time is defined as the constant *MAX_SEAT*.

A *Car* component can perform movement actions only when it is in one of the following states: *NORMAL, PRIVILEGED*. Cars in the state *NORMAL* have fewer passengers than the value of the constant *SEAT_THRESHOLD*. A car can change its state to *PRIVILEGED* by interacting with *QueuePoint* components in order to increase its number of passengers. Specifically, when a *Car* component is co-located with a *QueuePoint* component, the car and the queue can perform a sequence of actions in order to transfer a number of passengers from the queue into the car.

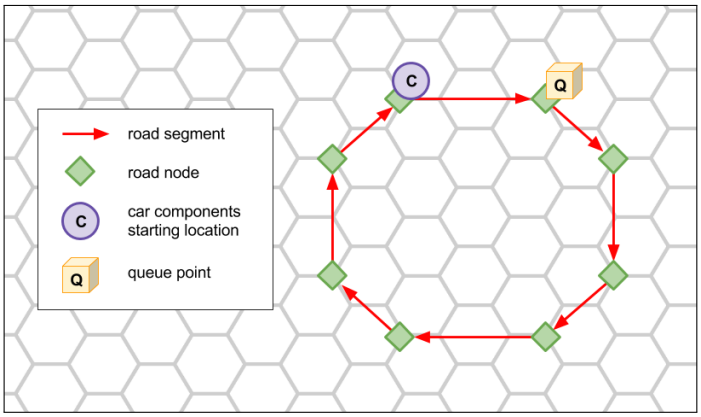

Fig. 4. A schematic view of the carpooling scenario. Path topology is the same for both the fast and the slow lane. The movement action over slow lanes has a lower rate.

QueuePoint components can be in one of the following states: *EMPTY, FULL, FILLED* and *OCCUPIED*. If the queue is not *EMPTY* or *OCCUPIED*, it can synchronize with the car on the *offerPerson* output unicast action. The *offerPerson* action sends a message from the *QueuePoint* component to the *Car* component containing information about the number of people waiting in the queue, available for pickup. *Car* components try to maximize the number of new passengers, while respecting the constraint that the number of uploaded passengers has to be less than or equal to the number of people available at the queue and the remaining capacity of the car. The *Car* component and the queue then perform the *carUpdate* unicast action in which the car informs the queue the number of passengers it can take, and the queue decreases its size accordingly, as shown in Figure 5.

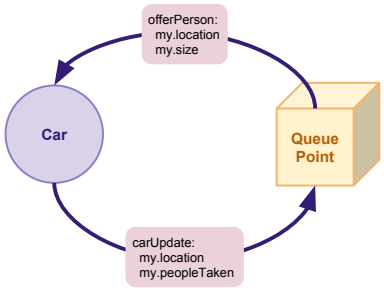

Fig. 5. A schematic representation of the information exchange between Car and Queue Point components during passenger pickup.

During this transfer, both the *Car* and the *QueuePoint* go into additional transition states. For a car, this state is *LOADING* and any car in this state cannot perform movement actions. For a queue, the state is *OCCUPIED*, which signifies that the component is busy performing a pickup action sequence with a car and cannot start performing pickup actions with any additional cars.

In this model, passengers waiting in queues do not have a specified destination or direction in which they want to go. It is assumed that if a person is waiting at a particular *QueuePoint*, they are willing to travel in the direction of cars that arrive at this *QueuePoint* to perform pickups.

To represent the completion of journeys, *Car* components also perform a spontaneous (unsynchronized) broadcast output action *releasePassenger*, with a constant rate, which decreases the number of passengers in the car by one. This is analogous to real world situations in which people queue at designated locations, but can get out at arbitrary times and locations.

6.2 Results

The results of simulation runs of our model are presented in Fig. 6 and Fig. 7. Rates have been chosen to appropriately represent the real world scenario. In order for the travel in the privileged lane to be beneficial, the movement action over this lane must have a higher rate than the movement action on the slow lane. The *offerPerson* action of the *QueuePoint* must also be sufficiently fast to ensure that the delay imposed by the interaction at the *QueuePoint* can be compensated by the increased speed in the priority lane. The rate at which a queue acquires new passengers is another value that can have an impact on the efficacy of the scheme. In real world scenarios, queues usually do not have a constant maximum size, but they do not grow infinitely; an approaching pedestrian will choose not to join the queue when it is sufficiently large. Similarly, a model in which the queue size is very low cannot demonstrate any benefit from carpooling.

7 Conclusions

In this paper we have presented a newly-developed software tool which assists with the creation of CARMA models of systems in which location and movement play a significant role. CAS by their nature are large-scale systems so concepts such as location, separation, distance and movement very often have roles to play in their models.

By concentrating on location and movement, our graphical modelling tool provides a convenient separation of concerns between the spatial aspects of a model (such as location, proximity and movement) and the dynamic aspects of a model (such as attribute and state update, communication, and synchronisation). We believe that this separation can be helpful in allowing the modeller to focus their attention on particular aspects of the model in isolation.

Our graphical model-generation tool handles all of the low-level aspects of location representation such as placement on a co-ordinate system and the consistent handling of co-ordinate values throughout the model. This level of detail

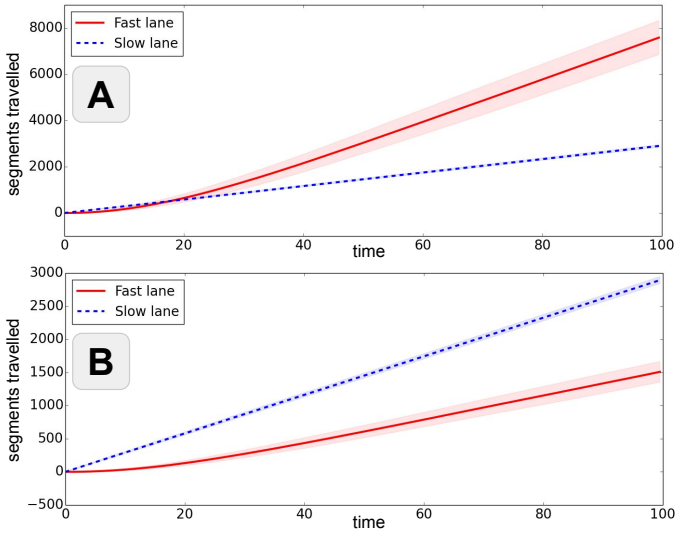

Fig. 6. Experiment showing how the movement rate in the fast lane impacts on lane usage (number of road segments traversed). Panel A: fast lane movement is 5 times faster than slow lane; Panel B: the movement rate is the same in both lanes.

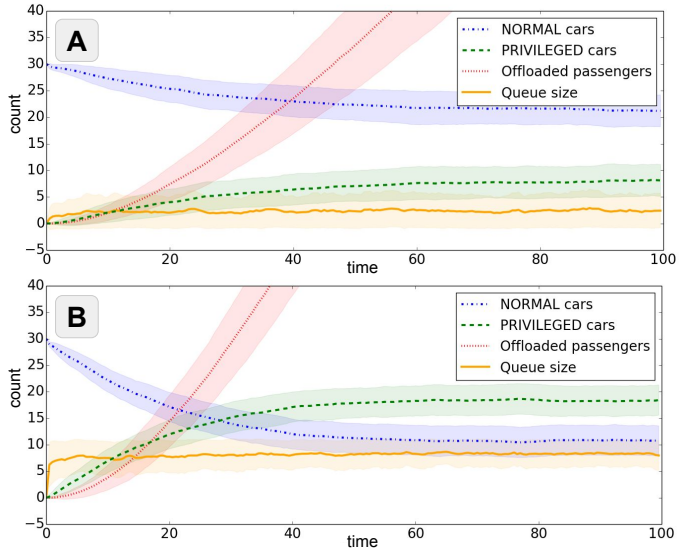

Fig. 7. Experiment showing the impact of the rate at which people are loaded as QueuePoints. Panel A: the loading rate is 1.5; Panel B: this rate is 10.0.

is often tedious and error-prone to maintain manually so we believe that the model generation approach also benefits modellers here.

We demonstrated the use of our software tool on a typical CAS case study and paired our model-generation tool with the CARMA Eclipse Plug-in to take a model of a car pooling system from high-level design through compilation into Java and subsequent execution as a simulation study of the system. This gave us insights into the dynamics of carpooling, and provides some validation of the correctness of the transformation of our graphical design into running code.

Acknowledgements: This work is supported by the EU QUANTICOL project, 600708. We thank the anonymous referees for many helpful suggestions.

References

[1] M. Loreti and J. Hillston, "Modelling and Analysis of Collective Adaptive Systems with CARMA and its Tools." To appear, 2016.

[2] Y. Abd Alrahman, R. De Nicola, and M. Loreti, "On the power of attribute-based communication," in *Formal Techniques for Distributed Objects, Components, and Systems: 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings* (E. Albert and I. Lanese, eds.), pp. 1–18, Springer, 2016.

[3] R. De Nicola, D. Latella, A. Lluch-Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, and A. Vandin, "The SCEL language: Design, implementation, verification," in *Software Engineering for Collective Autonomic Systems - The ASCENS Approach* (M. Wirsing, M. M. Hölzl, N. Koch, and P. Mayer, eds.), vol. 8998 of *Lecture Notes in Computer Science*, pp. 3–71, Springer, 2015.

[4] V. Galpin, "Modelling ambulance deployment with CARMA," in *Coordination Models and Languages: 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings* (A. Lluch Lafuente and J. Proença, eds.), pp. 121–137, Springer, 2016.

[5] J. Hillston and M. Loreti, "CARMA Eclipse plug-in: A tool supporting design and analysis of Collective Adaptive Systems." To appear, 2016.

[6] H. Yang and H.-J. Huang, "Carpooling and congestion pricing in a multilane highway with high-occupancy-vehicle lanes," *Transportation Research Part A: Policy and Practice*, vol. 33, no. 2, pp. 139 – 155, 1999.

[7] I. Hussain, L. Knapen, S. Galland, A.-U.-H. Yasar, T. Bellemans, D. Janssens, and G. Wets, "Agent-based simulation model for long-term carpooling: Effect of activity planning constraints," *Procedia Computer Science*, vol. 52, pp. 412 – 419, 2015.

[8] Y. Guo, G. Goncalves, and T. Hsu, "A multi-agent based self-adaptive genetic algorithm for the long-term car pooling problem," *Journal of Mathematical Modelling and Algorithms in Operations Research*, vol. 12, no. 1, pp. 45–66, 2012.

[9] G. Correia and J. Viegas, "A conceptual model for carpooling systems simulation," *Journal of Simulation*, vol. 3, pp. 61–68, 2009.